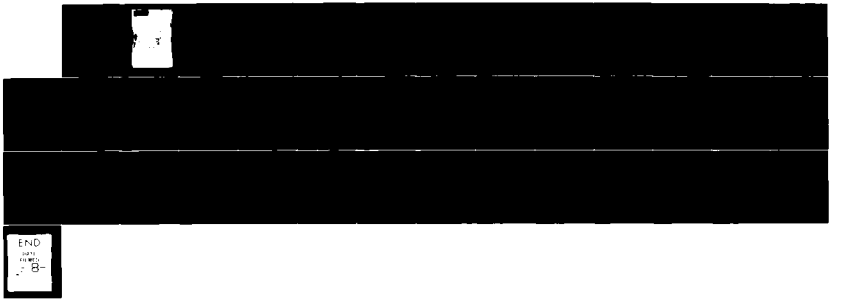
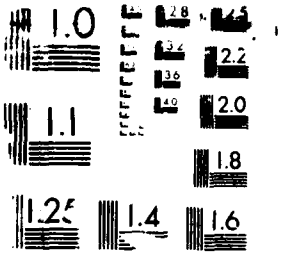
AD-A188 985 PUBLIC TOOL INTERFACES FOR SOFTWARE DEVELOPMENT ENVIRONMENTS(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) M STANLEY ET AL. SEP 87 RSRE-87004 DRIC-BR-104822 1/1

UNCLASSIFIED F/G 12/5 NL



END
DATE FILMED
-8-


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 87004

Title:
**Public Tool Interfaces for Software Development Environments**
Authors:
**M.Stanley, N.E.Peeling and I.F.Currie**
Date:
**September 1987**

**Summary:**
This report looks at the problems of defining a portable public tools interface to underpin the software tools needed for the development and maintenance of software systems. It proposes a requirement specification and gives the rationale behind the proposed requirements. The rationale also suggests some facilities and implementation features to enable the requirement to be met. It indicates that the requirement is not necessarily over ambitious.

RSRE REPORT 87004

**Public Tool Interfaces for Software Development Environments**

M.Stanley, N.E.Peeling and I.F.Currie

CONTENTS



(continued...)

CONTENTS (continued)

RSRE REPORT 87004

# Public Tool Interfaces for Software Development Environments

M.Stanley, N.E.Peeling and I.F.Currie

## 1. Introduction

1.1. There is a move in the computing community towards defining and producing a portable public tools interface (PTI) to support a wide range of tools needed for software development and maintenance. The notion of a standard PTI is attractive. Eventually it is hoped to agree an international standard PTI. Given a really good standard interface available on a wide variety of different machines, tool writers should be able to produce tools that use the interface, confident that their market was not restricted to specific hardware users. Furthermore, project managers should more easily be able to tailor their tool kit to the needs of the project.

1.2. A PTI is in essence an operating system without the utilities (tools) such as compilers, command line interpreters, editors, loaders and file management software that are usually deemed to be part of an operating system. The facilities of a PTI are provided as a set of procedure calls (in appropriate languages) that can be incorporated in user supplied tools. The PTI may include more facilities than a standard operating system. For example the interface may include a database system such as PCTE OMS (ref 4), not normally provided by an operating system. A tool is here deemed to be a program or set of programs, working on the interface, available for use by more than one user of the interface. Programs working on the interface but restricted to use by the program developer are deemed to be application programs.

## 2. Documentation for a PTI

2.1. Several initiatives are attempting to define the requirements of a PTI. They include (see glossary) the RAC (already available from DoD)(ref 3); the NRAC (a NATO equivalent of the RAC, from the NATO study group known as the TSGCEE SWG on Ada/APSE); the EURAC (a European equivalent of the RAC, from a combined NATO industry/military group called IEPG) and possibly others.

2.2. The documentation needed to support a PTI is shown schematically in Figure 1. In order to engender a common understanding among funding bodies, developers and users of a PTI about what a PTI really is, a short requirements document is needed. This will identify the mandatory and desired characteristics of a PTI. The requirement must not include statements on the way in which the objectives will be met nor identify the detailed facilities needed.

2.3. The requirements document should be supported by a rationale document. The rationale will indicate the reasons for the requirements. It will elucidate the underlying concepts. It will also

increase confidence that the requirements are realistic by including suggestions on facilities and implementation features which could enable the PTI to meet the requirements. The suggestions in the rationale would not be mandatory. They would expand on the requirement, explaining the assumptions underlying the requirement and the implications that follow from the requirement.

2.4. A separate facilities specification should define the facilities chosen to meet the requirement, such as transactions to support integrity, or an entity relationship database to support the expression of relationships and data types. The facilities specification may also address any non- functional features necessary to achieving the requirement, such as data representation. If it is decided, in the interests of economy or speed of implementation, that the requirement need not be met in full, then the facilities specification should indicate where the requirement has been relaxed. A rationale document justifying the choice of facilities to meet the requirement should support the facilities specification,

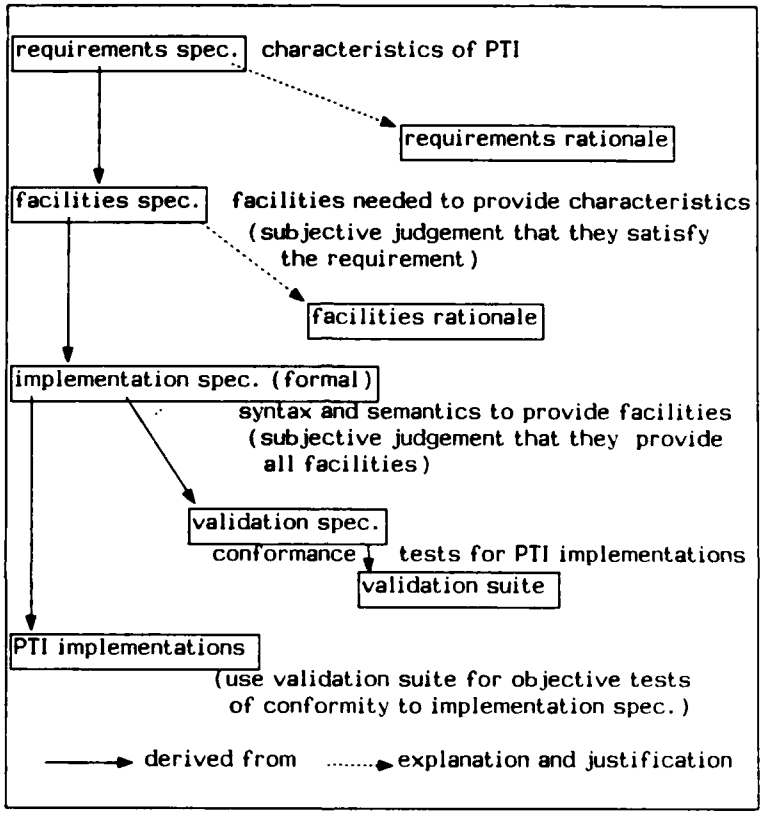
2.5. The facilities specification should be followed by an implementation specification. This will indicate the actual procedure calls, with syntax and semantics, needed to provide the facilities, plus any necessary detail of things like data representation to be used by all implementations.

2.6. The requirements of a PTI should be identified by considering the full range of needs of tool writers and tool users over a very wide range of different types of software tool. It is vital to avoid falling into the trap of specifying requirements by looking at facilities and working out what requirement they meet. It is also important to avoid restricting the requirements to those that can be met by facilities already available in some operating system, prototype PTI or existing PTI plus perhaps a minimum set of further facilities needed to satisfy the needs of specific user communities such as the military. Such an approach would, in the longer term, be counterproductive. A PTI would easily be produced which satisfied the stated requirements but which did not actually provide the features necessary to a successful PTI. Even if it is decided, for political or economic reasons, that it is not desirable in the short term to meet the full PTI requirement, the requirement will provide a useful yardstick against which PTI developments can be measured.

2.7. It is not envisaged that objective tests as to whether a PTI meets the requirement would be feasible. A PTI implementation would not be tested against the requirement. There would need to be subjective judgement on the extent to which the facilities specification matched up to the requirement. More precision would be possible in identifying the extent to which the implementation specification provided all the facilities in the facilities

specification. Validation of implementations could be made only against the implementation specification.

Figure 1 Documentation



## 3. A PTI requirements specification

The authors recognise the difficulty of writing a requirements specification. This section and the next section are an attempt to define the first two documents, the requirements specification and the rationale for the requirements. The requirements listed here are those identified as important by the authors both as a result of their own experience in developing and maintaining software and as a result of the research on software development environments carried out at RSRE. Work on the Flex PSE (ref 12) and the Ten15 abstract machine (ref 29), both developed by RSRE, has been particularly relevant. Other important influences have been the work on software development environments and on tool interfaces carried out under the ESPRIT programme (e.g. the development of PCTE (ref 4)), the requirements work sponsored by IEPG on the EURAC and related work in the CAD community (ref 9).

### 3.1. Mandatory and optional requirements

In this document 'shall' indicates a mandatory requirement, 'should' indicates a desirable but not mandatory requirement. Bracketed comments in this section belong more properly in the rationale but are included to make the requirement more readable.

### 3.2. Scope

The PTI shall provide facilities to support software tools for the development and maintenance of both large and small software systems, including development and maintenance of real-time software systems and systems which will run on a target machine which does not support the PTI. The PTI should support as wide a range of software systems as possible.

### 3.3. Partitioning

3.3.1. The PTI shall be partitioned in such a way that the partitions operate independently and may be understood independently. There shall be no undocumented dependencies between partitions. It shall be possible to install the PTI without those partitions that are not required by the tools to be supported at that installation.

3.3.2. If the PTI provides both high and low level facilities it shall police the use of any low level facilities to ensure that their use does not compromise the protection enforced by higher level facilities (e.g. preventing the use of filestore manipulations to destroy a database).

### 3.4. Completeness

3.4.1. The PTI shall provide a complete set of facilities such that most tools do not need to access facilities outside those of the PTI in order to carry out their function without compromising other requirements (such as performance and integrity). (It is desirable that any tool shall be implementable using the PTI.)

3.4.2. If the PTI provides high level facilities (such as a database) the lower level facilities needed to implement them shall also be accessible to tool writers.

3.4.3. The set of facilities shall be complete in the sense that PTI operations shall form a closure. (i.e. it shall not be possible to compose two or more PTI operations to create an operation which is not valid in the PTI. For example if the PTI were to impose a constraint on the number of files used by a single operation then an operation resulting from the composition of two operations might break the constraint.)

### 3.5. Portability of the PTI

The PTI shall be capable of implementation on a variety of different machines and system architectures, including bare machines and a variety of operating systems. The PTI shall be designed so that re-implementation on a different machine or system architecture, without compromising other requirements, involves the minimum of re-writing.

### 3.6. Tool Reuseability and Interworking

3.6.1. It shall be possible for a tool or application to accept arbitrarily complex data structures and to write arbitrarily complex data structures both to and from long term storage. It shall also be possible for a tool or application to communicate arbitrarily complex data structures to another tool without involving long term storage.

3.6.2. It shall be possible for the PTI to check that a tool is applied to or accepts data only if the data has the appropriate structure.

3.6.3. The data structures expressible in the PTI shall be compatible with the data types of the languages supported by the PTI.

3.6.4. Any tools whose interfaces match shall be composeable to create new tools. The resulting tools shall be composeable with other tools in the same way. The PTI shall be able to check that the interfaces between the tools match.

### 3.7. Database

3.7.1. The PTI shall provide sufficient facilities to support a database suitable for the efficient storage and manipulation of any granularity of data (fine or coarse grain) and in which relationships between data objects can be expressed.

3.7.2. A database shall be provided that permits expression of relationships between values. It shall support description of the structure of data objects held in long term storage which is compatible with the data structures provided for checking the correct use of tools.

### 3.8. Extensibility

3.8.1. It shall be possible to introduce new tools to work on the PTI without the need for privilege. Adding a tool to the environment means making the tool available for use by more than one user of the environment.

3.8.2. It shall be possible to compose PTI operations to create new operations efficiently without the need for privilege. It shall be possible to introduce new PTI operations as necessary.

### 3.9. Orthogonality

3.9.1. The PTI facilities shall be orthogonal. This implies the aim that the facilities shall provide a complete but minimal set of primitives which can easily be composed to provide higher level more application-specific features. It also implies the aim that the basic facilities shall be mutually independent (i.e. changes in the definition of one basic facility shall not imply changes to any other basic facility.) "Special cases" in the definition should be kept to an absolute minimum. All facilities shall be available in all contexts.

### 3.10. Homogeneity

The PTI implementation specification shall, as far as possible, provide a homogeneous interface in the sense that the way in which the interface procedures handle input and output parameters, delivery of results, data types and failures or exceptions shall follow defined conventions.

### 3.11. Performance

The PTI facilities shall not provide significantly worse performance than provided by existing operating systems. Use of the PTI shall not markedly degrade the performance of existing tool sets. It shall be possible to write new tool sets working on the PTI which exhibit similar performance to that which would be achieved using conventional operating systems.

### 3.12. Integrity

The PTI shall provide integrity in the sense that it will not permit normal operations on inconsistent data in filestore, mainstore or across a network. In the event that a failure might result in loss of consistency, the PTI shall ensure that the filestore is returnable to a consistent state (if necessary by returning to an earlier state).

### 3.13. Security

The PTI shall exhibit the integrity of filestore, of running programs and of the network necessary to implement a security policy, either military or civil.

### 3.14. Robustness.

The PTI shall be designed in such a way as to isolate, (as far as possible) errors and their consequences, to control errors and to recover from them where possible.

### 3.15. Portability of tools

It shall be possible to run any tool implemented on the PTI on any implementation of the PTI that provides the necessary hardware.

### 3.16. Interoperability

The PTI shall support the transfer of data and data structures from one implementation of the PTI to another.

### 3.17. User interface

The PTI shall provide facilities for tools to communicate with bit-mapped workstations, to support tools such as command line interpreters, editors and graphics tools.

### 3.18. Interactive working

The PTI shall provide facilities for the highly interactive use of tools in the sense that any tool can invoke any other in an unanticipated way without significant performance penalties.

### 3.19. Multi-tasking

An executable object (tool or procedure) that is being executed is called a process. A process comes into being when it is activated. The PTI shall provide facilities for processes to run concurrently. The facilities shall support synchronisation of concurrent processes and safe sharing of structured data without compromising integrity. The PTI shall provide facilities for mutual exclusion and for control and termination of processes both by the invoking process and by independent processes.

### 3.20. Mixed language working

The PTI shall be programming language independent. It shall permit the interworking of tools written in different languages. It shall also support mixed language working within tools (e.g. tools written in Ada but using C procedures and vice versa) to the extent that the types within the languages are compatible.

### 3.21. Uniformity

The PTI shall provide a uniform, language independent representation of data. (For example so that a debugger can be developed to work with procedures in any language).

### 3.22. Distribution.

The PTI shall be capable of working across a distributed network of dissimilar workstations, mainframes and file servers. It shall

provide facilities to enable integrity across the distributed network to be maintained. It should be possible to implement these facilities without allowing failure of a part of the network to deny service to any other part of the network which does not directly access the part that has failed. (This implies a mechanism for performing remote actions, facilities to support a distributed commit across the network and facilities for a resilient lock mechanism across the network).

### 3.23. Established technology

The PTI shall contain only features which have been demonstrated in existing commercial or military software systems.

### 3.24. Technological compatibility

The PTI shall adopt existing standards where applicable.

### 3.25. PTI validation

There shall be a formal definition of the implementation specification of the PTI. Implementations of the PTI should be validated using procedures derived from the formal specification.

Figure 2 A PTI based PSE- the layers.

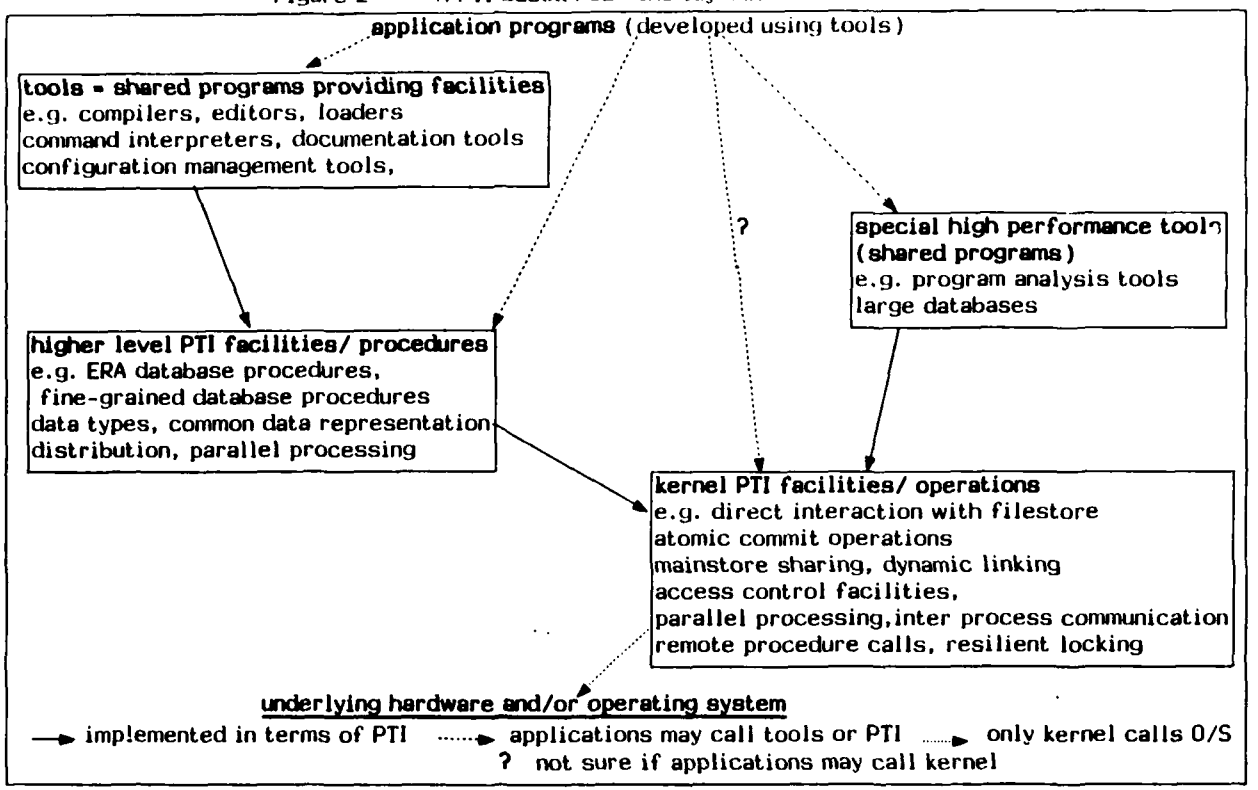



Figure 2 A PTI based PSE - the layers

## 4. Rationale for the requirement

This section gives the rationale for each individual PTI requirement, preceded by a discussion on the different levels of facility in a PTI. It ends with suggestions on the lower level facilities needed to support the requirements.

### 4.1. Different levels of facility

4.1.1. Although some of the requirements for the PTI imply the provision of high level facilities such as a database, it will be shown that others imply the accessibility of a low level kernel.

4.1.2. A kernel of low level facilities is necessary to satisfy the requirement to use only established technology. The experimental work needed to demonstrate the viability of a software support system that provides only high level facilities has not been done. Work is underway to implement some tools on PCTE (the PACT project) but care is needed in evaluating the results. Unless tools expected to be difficult to implement on PCTE are included in the project, (e.g. Ella, see the rationale for the database), successful tool implementation may still fail to provide the required demonstration.

4.1.3. If a database or some similar high level interface is the lowest level of interface provided, it is likely that we will eventually find some flaw that we have not yet identified and that cannot be circumvented without a lower level kernel. No set of higher level facilities can be guaranteed to provide for all the needs of tool writers. Any essential facility omitted from the higher levels must be implementable in terms of kernel facilities should the need arise.

4.1.4. As well as being necessary for the completeness of a PTI, the need for a kernel is a unifying thread that solves many of the problems in other requirements. This section is included to demonstrate the power of this unifying concept even though it trespasses on many of the following sections.

4.1.5. While accessible low level facilities are essential to a PTI kernel they do not preclude provision of a variety of higher level facilities, such as a database, implemented in terms of the kernel. The PTI kernel should be the minimum set of primitives needed to run the system and to allow efficient and safe interaction with filestore, with other processes and with other devices. Most tool writers would probably choose to use only the higher levels, with the additional protection and facilities provided.

4.1.6. It is vital for the health of the PTI that the set of kernel facilities be correctly identified and carefully implemented. Higher level facilities may be added as the PTI evolves but the full set of kernel facilities is needed to facilitate development of the higher levels as well as for tool development. The last subsection in this

rationale discusses the choice of kernel facilities. Much of the rest of the rationale is concerned with higher level facilities.

4.1.7. It might be argued that the provision of kernel facilities that allow a user to bypass the higher level facilities such as the database will make security and integrity provision harder. For example, the database interface will no longer have control of filestore allocation and access control. However, the kernel might still police the filestore allocation while giving users the right to request specific allocation. If the regions of filestore to which a tool writer has direct access were either disjoint from regions used by the higher level facilities or wholly contained within some higher level structure then there need be no loss of integrity and security. Against the risk of diminished security resulting from bypassing higher level PTI facilities must be set the consideration that, should tool writers find it impossible to implement high integrity, high performance tools using the higher levels and should they also be denied the necessary facilities at the kernel level then they will bypass the PTI entirely. They may be driven to implementing the tools on the underlying operating system thus preventing any part of the PTI from controlling or even checking their use of filestore or other system facilities.

4.1.8. It has been argued that the protection of filestore accessed by high level facilities from corruption by the kernel will prevent *intertool communication.* It is accepted that tools written using the kernel may not directly share data with tools implemented using database facilities. It will, however, be possible to write tools to transform the results of low level operations to a form suitable for insertion in the database. Other tools might take data from the database and provide it to tools that use low level facilities.

4.1.9. It has also been argued that a kernel of low level facilities must hinder tool portability. There is no reason why most of the low level facilities should not have the same syntax and semantics on the various different machine architectures. Most machine architectures provide essentially similar facilities for mapping onto the hardware. Machine dependencies can be encapsulated as constants supplied as parameters. While it is always possible for programmers to write non-portable software, the skilled tool writer (an unskilled programmer will probably not choose to use kernel facilities) will ensure the portability of his tools by the use of parameters for all machine dependent constants.

## 4.2. Scope

4.2.1. The main aim of a PTI is the cheap provision of as wide a range of software tools as possible. A PTI suitable for the support of software tools must provide facilities for calling tools from other

tools, and for communication between tools and between a tool and the underlying computing hardware.

4.2.2. While the PTI must provide facilities for communication between tools and users, it is not clear whether login and logout or any standard way (command language or menu) for users to call tools or to give commands to the operating system, should be part of a PTI. It would be too constraining to define a single command mechanism but guidelines indicating a recommended method of connecting to a PTI based system, with the form of access controls to be imposed etc. are probably advisable.

4.2.3. The facilities specification needs to consider what extra facilities, if any, will be needed for host/target software development, for example to allow a developer to monitor processes running in a target which may not support the PTI.

### 4.3. Partitioning

Partitioning the PTI and implementing it as a kernel plus a set of higher level facilities is not subsetting. All the PTI facilities, including all the higher level facilities, would be mandatory for conformance, in the sense that every PTI implementation must provide them. A partition would be optional only in the sense that the tool writer could use a PTI without those parts he does not need, secure in the knowledge that all the PTI packages he chooses to use will still work. If the partitions are independent an individual installation of the PTI might choose not to provide a partition (such as a database 4.3.1. package) that is not needed by the tools to be installed. The software would still exist and could be provided if needed at a later date. For example, an installation on a single processor would not need to include facilities for distribution.

### 4.4. Completeness

4.4.1. The PTI needs to be complete in the sense that no necessary facilities are omitted. A PTI which did not provide the full range of facilities might have to coexist with tools which bypass it, with all the problems that result from coexistence.

4.4.2. Current operating systems are an established technology providing a kernel of low level facilities from which the full range of higher levels can be built. As already indicated, the paucity of experience of PTIs based only on higher level facilities makes it improbable that all necessary facilities will be remembered at the higher levels. If a forgotten facility may only implementable in terms of the high level facilities by compomising other requirements such as efficiency. Access to low level facilities will then be essential. Users will be tempted to circumvent a PTI that fails to provide the flexibility and level of facilities needed to enable tool writers to provide performance, integrity, security and data checking. (This

may well involve facilities for tailoring a tool for specific hardware or operating systems. )

4.4.3. A major objective for a PTI is the cheap provision of tools working on the interface. Conversion of existing tools to work on the PTI is therefore desirable. This supports the argument for a low level kernel. Unless the PTI provides facilities similar to those used in the current tool interface to the operating system on which the tool runs, tool conversion could involve a radical redesign to fit the tool to the different form of interface. It could prove difficult or costly to convert existing useful tools to work on a new form of interface. Conversion of existing tools to a PTI may degrade performance or integrity to an unacceptable degree unless an appropriate set of low level facilities, such as provided by current operating systems, is accessible.

4.4.4. A PTI needs to be capable of growth if it is to adapt to changing software development methods and ideas. It should allow the addition of new operations as required. If the PTI operations do not form a closure it will be possible to violate the rules for the PTI by combining operations each of which is individually valid. The flexibility and orthogonality needed to encourage growth will be lost. It will only be safe to combine operations in certain contexts. Users will hesitate to combine operations if the resulting operation might fail.

## 4.5. Portability of the PTI

### 4.5.1. Underlying operating system

4.5.1.1. PTI implementations need to be relatively portable to ensure wide availability without incurring undue expense.

4.5.1.2. A PTI may be implemented either on a bare machine or on top of a basic underlying operating system or on top of an operating system supplemented by some facilities implemented on the bare machine. If the PTI is implemented on top of an operating system then the operating system must provide all the basic facilities defined for the PTI. Any facilities that cannot be provided directly as operating system facilities must be provided indirectly in terms of operating system facilities. Should the underlying operating system be unable to provide directly or indirectly for some PTI facility then that facility must be implemented on the bare machine, (bypassing the operating system). For example, if the PTI is to be implemented on an operating system (such as Unix) which does not provide facilities for preserving the necessary levels of performance and integrity in the use of filestore, nor for dynamic linking (refs 19, 20 and 21) then the operating system will need extension. The quality of a PTI implemented on top of an operating system, and the quality of the tools it supports, is bounded by the quality of the underlying operating system. All operating system facilities used to support the

PTI must therefore exhibit the required integrity and performance. The operating system must be bypassed or extended in those cases where the operating system falls short.

4.5.1.3. A PTI should be portable in the sense that it should be possible to move implementations of the PTI to different computer hardware or to different underlying operating systems with a minimum of effort. If a PTI were fully implementable in terms of the primitives of a portable operating system then portability of the PTI to different computer hardware could in a sense be achieved by portability of the underlying operating system. Such a PTI will not be fully portable in that it will not necessarily be feasible to move it to other operating systems except by providing an additional layer in which the specified operating system primitives are implemented on top of the host operating system, which would probably impact performance.

4.5.1.4. Implementation of a relatively portable PTI on a bare machine would be possible by isolating the machine dependent parts and by the use of parameters for machine dependent features such as disc sector size.

4.5.1.5. In order that the higher level facilities of a PTI be portable they should be implemented in terms of the kernel. The kernel facilities should be specified and designed to be as independent as possible of any specific operating system characteristics. A kernel that relies on characteristics peculiar to a particular operating system will be more difficult to move to other operating systems.

4.5.1.6. Too close an association between the PTI specification and a particular underlying operating system can also give rise to a problem that has nothing to do with portability. The problem is control of the PTI specification. The specification of a PTI defined in terms of specific operating system primitives may change as a result of upgrades to the operating system. This could lead to problems because the PTI specification could effectively be changed without warning by the operating system suppliers.

### 4.5.2. Unix

4.5.2.1. The wide availability and easy portability of Unix would clearly enhance the availability of any PTI based on Unix. It is therefore worth considering the desirability of specifying a PTI suitable for implementation on top of Unix.

4.5.2.2. A PTI defined for easy implementation on any specific operating system will probably be inefficient when implemented on other operating systems. Unix is no exception. Similarly, it will be difficult to control the specification of a PTI whose definition is too closely associated with a specific operating system. Unix is again no

exception. Should the PTI definition be closely associated with Unix it will be necessary to sort out consequences of changes to Unix definitions and of the development of Posix (a Unix based operating system being promoted in the USA as a possible Unix standard).

4.5.2.3. Unix is not an ideal operating system on which to implement a PTI. It was created as a laboratory research vehicle. It was not designed as a commercial operating system. It has recognised shortcomings that have been widely reported (refs 1,2,5,6 and 13). Any PTI implemented on an operating system with these shortcomings may provide all specified functions but will fail to satisfy other PTI requirements such as performance, security or integrity. The authors of Unix do not believe it is possible to add the required characteristics as a bolt on extra (refs 1 and 2 and 5).

4.5.2.4. Many of the problems associated with Unix are probably a result of specific implementation features rather than being inherent in the specification of Unix (ref 6). Other problems are associated with the lack in Unix of those low level facilities need to support dynamic linking and those necessary to enable interactions with filestore to be provided with integrity and performance. It is important to consider the minimal set of changes and extensions (including possible changes to the implementations) necessary to make Unix an acceptable underlying operating system for a PTI.

4.5.2.5. In addition to the extensions needed to augment Unix with low level facilities, possible extensions to enable the undesirable features of Unix to be bypassed (for example bypassing the cache when necessary) should be considered.

4.5.2.6. A Unix look-alike reimplemented to solve the problems properly by avoiding the specific implementation features which make Unix undesirable could probably be achieved. However, a Unix look-alike which did not do things such as filestore handling in the same way as the AT&T software is unlikely to be recognised as 'proper' Unix. Problems might be experienced in importing Unix tools if the semantics of the operating system changed as a result of reimplementation. Reimplementation of Unix may well prove uneconomic. The kernel extensions would still be needed.

4.5.2.7. Bearing in mind the above comments, some evaluation should be made of the cost effectiveness of a PTI based on a modified and extended Unix.

## 4.6. Tool Reuseability and Interworking

4.6.1. The requirement for tool interworking and safe tool composition implies support for data structure definition by the PTI and for long term storage of structured data on filestore. Data

structure definition is appropriate to the higher levels of a PTI (e.g. a database). It is not applicable at the kernel level.

4.6.2. A PTI aimed at cheap provision of tools should encourage users to fit existing tools together in new ways to achieve new objectives. If primitive tools are expected to work correctly together to provide a more complex function, it is vital that any interface between tools (i.e the data provided by one tool to another) be understood in the same way by both tools. If the data output by tool A will be misinterpreted if input to tool B, then it should not be possible to compose tool A with tool B. Most current operating systems support only unsafe tool composition in that there are no safeguards that prevent a receiving tool from picking up data with implicit structure, A, and assuming a totally different implicit structure, B, with unpredictable results. Tools and applications are expected to make their own interpretation of the hidden structure in a file or pipe. For example, a tool may output a byte stream with implicit structure A:(boolean,integer, date, real number, character string) which will be input to another tool in the set. A third tool may expect a byte stream with implicit structure B:(integer, date, date, boolean, character string). If a tool picks up structure A when expecting structure B, it will misinterpret the data with unpredictable and sometimes unnoticed consequences.

4.6.3. Data structuring provides facilities to define and to use data structures not only within individual programs but for communication between programs and for long term retention of data. A PTI should, at the higher levels, permit a user to define the structure of the bulk of the data associated with an object such as a file, pipe or message so that the structure can be checked by the receiving tool. A tool cannot check the structure of data that has no explicit structure. If implicitly structured data is transmitted from tool to tool via the unstructured content of files, messages and pipes, it can be shared correctly only if the tools are written with foreknowledge of the structure.

4.6.4. The set of basic value types provided in the higher levels of a PTI must include all types necessary to normal data usage (refs 18 and 24), including real numbers. It is not, however, sufficient to define value types only for the more primitive values (attributes) associated with an object. The data typing system must also have the flexibility to allow tool writers to define arbitrary data structures to allow definition and use of the internal structure of data associated with bulk objects (such as file content). Users must therefore be able to compose their own value types (whose structures are compositions of a set of basic types) using type constructors such as vectors, references, cartesian product etc. Flexible use of arbitrarily small user defined data structures requires facilities to support fine granularity. (Note: abstract data types, where a type is defined not only by its structure but also by

the operations that can be performed on it are discussed under security.)

4.6.5. A value type should always have the same set of valid operations whether it appears on its own or as a part of some larger more complex structure. For example the operations on a boolean value should be the same whether the value is an element in a vector of booleans or a value on its own. The higher levels of the PTI should cause an exception or a program failure if an invalid operation is invoked on a value type (for example operating on a boolean field as if it were an integer). It is not sufficient to provide type information without also providing facilities to ensure that a tool writer does not ignore the information, for example using a date as an integer and performing arithmetic on it.

4.6.6. A tool writer should not only be able to define an arbitrary structure for data written to filestore or sent in messages or pipes to other tools, he should also be able to return (to the calling tool) an arbitrarily complex structure as a result of calling the tool or to pass an arbitrarily complex structure to another tool running concurrently. If results of tool execution are restricted to simple structures such as status values then filestore may have to be used unnecessarily to pass data from one tool to another. A tool returning an arbitrary data structure should be composeable with any other tool that requires that structure of data as input, without the performance penalty of writing the structure to filestore or into a message. There should be no need to predefine what tool will operate on the data next.

4.6.7. Not only should the PTI higher levels retain information on the structure of data but they should also aim to prevent programs or PTI operations from operating on values, objects or relationships of the wrong type. If a tool is to be prevented from operating on a wrong data type then the data type of the input and output parameters of a tool must explicitly be associated with the tool itself. An executable object in the PTI must therefore define the data type of the parameters of the executable object and of the results, so that type checking of program calls can be performed. Programs called on the wrong parameter type should fail. The data structures expressible in the PTI therefore need to be compatible with the data types of the languages supported by the PTI.

### 4.7. Database

4.7.1. It is commonly held that a PTI should include some form of database to enable the expression of relationships between objects. This would clearly be useful both for configuration management (for example identifying related components and versions of a product) and for tools such as program analysis tools needing to express relationships between fine grained objects. Note that the requirement

does not call for the database to be distributed. This is because the requirement for a distributed database would be at odds with a requirement to use only established technology. The authors are not aware of any existing military or commercial distributed database possessing both the required performance and the required integrity.

4.7.2. The form which a database supporting a PTI should take is not entirely clear. No single form of database can be appropriate for all tools. For example, it would be difficult to implement the database kernel of a tool such as Ella (refs 7 and 8) using only high level facilities such as PCTE1.4 OMS without destroying Ella's claims to both high integrity and high performance. (Ella on BSD4.2 Unix requires use of the fsync function to clear the buffers. It is difficult to see how to manage without it.) There is a valid argument for defining, as part of the PTI, a variety of different higher level interface sets appropriate to different types of tool. Provided that the PTI kernel provides sufficient low level facilities for introduction of alternative styles of database it is relatively unimportant that the selected style of database may not be an appropriate data model for some tools. The aim should be to select a form or forms of database which will adequately support a wide variety of tools.

4.7.3. An entity relationship database has been suggested as the most appropriate model for a wide range of tools. This may conflict with the requirement to use only a technology that has actually been used in real systems. Although entity relationship databases do exist the authors know of none that have been shown to exhibit the necessary integrity, performance and portability.

4.7.4. A database based on files as the unit of granularity will not be suitable for tools needing efficient access to fine grained data. Such tools include the Adam database (ref 7), any fine-grained database or persistent heap, tools to do with program analysis, verification and theorem proving, which need to represent expressions on filestore, often with links to other small structures, details of variables, parts of specifications etc. A fine grained data store would also be useful for program sharing, where the separately compiled units to be shared may be very small.

4.7.5. One application for which a PTI must provide support is configuration management. The database facilities selected should therefore be suitable for implementation of a configuration management system. To increase confidence that the correct facilities had been selected the rationale for the facilities specification should perhaps include an illustrated example of how the proposed facilities might be used. They might, for example, show how the proposed facilities could ensure that, given a program or a compiled unit, it would always be possible to reach the source text

from which it derived, with no possibility of ever mistakenly accessing an earlier or later version of the source text.

4.7.6. Since any software development system must expect to create quite a large amount of transient filestore values the PTI will need to provide facilities for filestore garbage collection. This is a very difficult problem if the PTI supports a cyclic database scanning several distinct filestore volumes.

## 4.8. Extensibility

### 4.8.1. New tools and privilege

4.8.1.1. Adding a new tool to an environment based upon the PTI involves making the tool available for use by more than one user of the environment. Unless it is easy to do this the toolset will not grow freely. If privilege is needed to add a new tool then there will be a temptation to distribute privilege more widely than would be compatible with the enforcement of security.

4.8.1.2. It is desirable that multiple invocations of a tool shall not cause multiple copies of the executable code to be loaded. This is particularly important if the tool code is large or is likely to be widely used. It should therefore be possible, without privilege, to make the executable code of a tool re-entrant and shareable between different users

### 4.8.2. New PTI operations

4.8.2.1. A system that cannot evolve will not continue in use. It must be possible to augment the PTI, evolving higher levels of PTI facilities after the initial definition and implementation. To evolve new facilities it will be necessary to compose existing operations to create new ones without unnecessary performance penalties. Tool writers will wish to create new operations from existing PTI operations or facilities.

4.8.2.2. For efficient composition of PTI operations the higher levels of the PTI should include 'operation' as a data type, with the data type of its input and output parameters. The set of PTI operations should not be totally defined by the set of procedure calls provided. If there is no way within the PTI of describing an object which is an operation then the sequence of existing operations which define a new operation must be interpreted rather than compiled. One cannot take advantage of the a priori knowledge of the structure of an object derived from an operation when applying the subsequent operation. One must check the structure again on input to each operation in the sequence. This has really dire effects on efficiency - the difference between interpretation and compilation in normal languages is a good analogy.

4.8.2.3. It may be that a need for a new PTI operation that cannot be

created by composition of existing operations will be recognised. It must be possible to augment the PTI to include such an operation. It is not possible to be certain that all necessary operations will be identified at the outset.

## 4.9. Orthogonality

4.9.1. If the primitive operations provided by the PTI are complete and if PTI operations can be efficiently composed then there is no reason why the PTI cannot be extended to support higher level facilities as the need arises and the technology develops. The basic set of facilities should be minimal and orthogonal (mutually independent) in order to to assist the designer in keeping the basic PTI simple, small and relatively easy to develop and to maintain. Any departure from orthogonality will give problems in maintenance as well as in use.

4.9.2. If certain facilities are available only in appropriate contexts then the PTI will be difficult to use. Tool writers will need to remember different rules for correct use of each operation and will therefore be more liable to make mistakes. It will be more difficult to provide tools to enforce or assist correct useage of operations if the rules vary according to context.

## 4.10. Homogeneity

If a homogeneous interface is provided in which all PTI procedures adopt similar conventions (as far as the supported languages allow) then it will be easier for tool writers to learn and to apply the procedures correctly in a variety of contexts.

## 4.11. Performance

4.11.1. Ref 14 indicates the importance of performance in a PTI. If the performance of a set of tools working on a PTI is significantly worse than the performance of similar tools provided without using the PTI then the PTI tools might be rejected by those parts of the user community for whom performance is a high priority. Performance is a difficult issue. Degraded performance of an individual tool may be acceptable provided the overall performance of the tool user in achieving his objective is not impaired. However if individual tools are perceived to give significantly worse performance than comparable tools in other environments it is likely that the overall performance will be deemed unacceptable.

4.11.2. There may be difficulties with the performance of high level facilities arising from the specification of the facilities themselves. Tools based on the higher level PTI facilities such as a database may be prepared to trade performance in order to use improved facilities but tool writers must be permitted to forego the additional facilities in order to achieve a better performance.

4.11.3. A tool writer needs a good mapping to the underlying machine architecture if he is to provide high performance tools. This confirms the necessity of a kernel of low level facilities. (Experience with standard interfaces in the CAD field supports this view, see ref 9. While a software development PTI may not be intended to support CAD tools we should learn from the experience of the CAD community in attempting their own PTI and from the problems they encountered, particularly with respect to the implementation of large databases.)

## 4.12. Integrity

4.12.1. Integrity is breached if incorrect operation of the system due to a software or a hardware failure results in users being given access to values to which they have no right or to incorrect values either in mainstore or filestore. Filestore integrity is breached if a system or software crash results in corruption of the filestore or database so that users are presented either with no value for an existing file or database object or (worse) with a wrong value. This is undesirable, and can be disastrous in critical applications.

4.12.2. Of particular concern is the integrity of the filestore following either a hardware or a software failure. Integrity of filestore data manipulated by the low level PTI facilities must be the responsibility of the tool writer. The low level kernel must therefore provide sufficient facilities to enable integrity to be enforced.

4.12.3. The low level facilities must not be able to compromise the protection provided by higher level facilities. Users of high level PTI facilities have a right to expect that the integrity of the database or filestore will be maintained without the need for individual applications to check the integrity of the objects and their relationships. The higher levels of the PTI should not take the risk of relying on tool writers either to protect the integrity of filestore or to minimise the risk of filestore corruption. A tool writer who chooses to ignore his responsibility for the integrity of his data might otherwise cause corruption of data needed by other tools. Any filestore corruption that can occur should be detected and recovered from. The PTI facilities specification needs to indicate how integrity will be maintained by the higher level facilities in the event of a hardware or software failure, how loss of integrity will be detected and how recovery will be achieved. It is also important to specify how the PTI will prevent use of kernel facilities from resulting in corruption of high level data. As far as is feasible, higher level objects should be inaccessible using the kernel facilities. This should be enforced.

4.12.4. A number of established technologies exist for implementing filestore integrity. The usual solution is to use a 'un-do/re-do' log.

This provides integrity facilities with an inevitable, but usually acceptable, decrease in performance, since every filestore update must be accompanied by a log update, also on filestore. Alternatively, tools may copy everything before starting a set of transactions, work on the copy and finally rename it as a unitary operation. This is of dubious value with large databases. (It was tolerable in SDS (refs 16 and 17) and on George 3 for up to about 2 megabytes.) Another possible solution to the problem of filestore integrity is to make the filestore non-overwriting except for the facility to overwrite a root pointer as an atomic operation (as in Flex (refs 11 and 12) and Adam (ref 7)).

## 4.13. Security

4.13.1. Security is breached if a user can illegally access or make unauthorised use of anything on filestore, in mainstore or across a network.

4.13.2. Access controls are a means of controlling and policing access. They need to be sufficiently flexible to permit sharing of objects between users and between tools where desirable without weakening the ability to exclude access to objects where necessary. It is probably the case that any operating system needs a very restricted set of highly privileged operations which may enable the user to violate normal access controls. However, it is vital for security that access to privileged operations be properly restricted. The PTI should not rely on access controls provided by an operating system which does not adequately prevent access to a privileged state.

4.13.3. If operations which a user may legitimately wish to perform (for example making code shareable between different executable programs) can be achieved only from a privileged state then privilege will be more widely distributed than is advisable, with consequential reduction in system security. Any tool writer may wish to make his code shareable by other users or tools. The PTI should therefore aim to provide for code sharing without the need for privilege.

4.13.4. Abstract data typing (i.e. definition of an object type by the set of programs or operations that can operate it (refs 10 and 24)) assists in the provision of security. Access to an object of a given type can be policed by the operations and programs that can operate on the object type. Any operations applied to the type but not defined for it should fail. Tool writers using the higher level PTI facilities should be able to define new abstract data types, together with the set of programs that can access them. (For flexibility there must also be an operation on the abstract data type to enable a user to extend the set of valid operations for the type.) The same set of operations should be associated with the abstract data type whether it appears on its own or as a part of some larger more complex

structure. For example the operations on a stack should be the same whether the stack is an object on its own or part of a larger object. It should be possible to define identical structures as distinct types, possibly associated with different operations and programs.

4.13.5. Integrity and security are closely linked. Abstract data types are very useful not only for security but also for integrity. Integrity is an essential prerequisite in the provision of security since corruption of data may invalidate any access controls.

## 4.14. Robustness.

4.14.1. A PTI designed to isolate errors and their consequences needs to detect incorrect use of the facilities as early as possible. Enforcement of correct use of data types by including the type of the parameters of operations and of tools as part of the tool or operation will assist in this. A uniform method of dealing with program failure will also assist. Partitioning the facilities will help to prevent accidental communication between different parts of the PTI.

4.14.2. It is important not only that any attempt to use PTI operations or tools incorrectly should fail, but also that the program or tool which invoked a failed operation should be unable to proceed in ignorance of the failure. Ignoring a failure should be a conscious decision on the part of the tool writer. It should not be possible to ignore a failure simply by omission of a status check. Failed PTI operations should not rely on return of a status report to indicate failure.

4.14.3. Tools to handle program failure, either in tools or in application programs, are essential for software development and maintenance. The facilities specification of the PTI needs to discuss how failures will be handled and how diagnostic tools will gain access to information about program failures. A common method of handling failures would remove the need for different debuggers for each language. The facilities specification should indicate whether handling of program failures will be the province of individual implementors of tools such as the compilers and loaders (and therefore language dependent) or whether a common method of handling program failures will be supported, shareable by all languages. A common data representation would assist in the provision of a common method for handling failures.

## 4.15. Portability of tools

4.15.1. In order that tools shall be portable between implementations of the PTI it is vital that there are no ambiguities in the specification nor gaps which would permit different valid implementations to behave differently or an implementation to behave differently if supported by a different underlying machine. This

re-enforces the importance of a validation procedure backed up by a formal implementation specification of the PTI.

4.15.2. It must be possible to ensure that a tool or application can be run only if the required facilities (e.g. screen size, graphics facilities, communications facilities, store size etc) are available. Facilities are therefore needed to interrogate the machine on which the PTI is working to discover what is available and to select appropriate parameter values (such as disc sector size) to tailor the tool to the underlying hardware. It would probably be useful to be able retain information as to the needs of a tool as part of an executable object.

4.15.3. One obstacle to tool portability will be implementation of real numbers, and of integers unless the number of bits in an integer is specified. A common data representation would assist in tool portability.

## 4.16. Interoperability

It is desirable not only that tools be portable between PTI implementations but that data should also be transferable. This implies not only the transfer of individual data items but also the preservation of relationships between the items, and the preservation of any structural and data type information associated with the items. This is similar to the requirement that data be transferable between dissimilar workstations on a network supported by the PTI. A common form of data representation supported by any necessary tranformation tools to deal with differences in word length is probably essential if this requirement is to be met.

## 4.17. User interface

Users need to interact with tools. Facilities for user interaction must to be part of the PTI to ensure that tools that interact with the user are portable to different user terminals and different PTI implementations. More research is needed on precisely what facilities are required for the user interface and what kinds of terminals should be supported. Users will expect to interact with tools through bit-mapped workstations and facilities are also needed for graphics tools. It may be difficult in this area to find an established technology. There are as yet no recognised standards for user interfaces although X-windows (ref 15) may be an evolving standard.

## 4.18. Interactive working

4.18.1. The PTI is required to support the highly interactive use of tools in the sense that any tool can invoke any other in an unanticipated way without significant performance penalties. There is considerable evidence of high user acceptance of interactive programming support environments in which the user can arbitrarily

choose to connect different tools together as dictated by the problem on which he is working. Flexible work patterns improve programmmer performance. The popularity of Smalltalk and of LISP environments is in part due to the highly interactive and flexible user interface. Both these environments are more than usually interactive.

4.18.2. Consider a user who calls a tool that delivers a result. The user may then wish to call some arbitrary tool to enable him to examine or process the result in some unanticipated way. He does not necessarily wish to have the result, which may be of only passing interest, automatically written to filestore for input to the next tool. Writing temporary data to filestore imposes performance penalties and consumes filestore unnecessarily. Unanticipated use of tools implies the ability to pass results to some arbitrary tool without the unnecessary degradation in performance caused by writing shared data to filestore.

4.18.3. Suppliers of tools such as command line interpreters cannot be expected to anticipate all the ways in which users will wish to call tools from one another. Facilities are needed to allow tools to call other tools as selected at run time by the tool user. Dynamic linking is essential if command line interpreters are to support unanticipated use of tools and inclusion of new tools as they become available, without the performance penalty of spawning a new process for every tool call. It is also important for efficient use of mainstore and for high performance tools. Dynamic linking is the ability to load modules at run time. Dynamic linking is probably a major factor in the popularity of the MAINSAIL programming language(refs 19 and 31). Systems without dynamic linking form a program into a single executable image in which all tools that can be directly invoked (i.e without spawning a new process) are linked into the program image before the program can run. Other tools may be invoked by spawning a separate process but spawning is an inefficient method of unanticipated tool invocation compared with direct tool invocation. Performance is often poor because the system has to copy the complete image when spawning a new process. It is usually necessary to use virtual memory and memory mapping to share code, which may slow down execution. Code can often be made shareable by a superuser, but this reduces security by invoking privilege when it should not be necessary. It is not usually possible for a spawned process to share mainstore data with the spawning process other than very simple data such as status flags or integer values.

## 4.19. Multi-tasking

4.19.1. The ability to run processes concurrently, to control concurrent processes and to provide for communication between them is necessary for the conversion of some existing tools to the PTI (for example, those that communicate with slow peripherals).

Concurrent processing and support for asynchronous communication between processes is needed for efficient communication with slow peripheral devices and for supporting tool interaction. It is essential that processes running concurrently but sharing data have adequate facilities for locking the data if integrity is to be maintained. The rules for control and termination of processes and for relationships between an activating and an activated process must be defined. It should be possible to initiate a concurrent process that is still dependent on the initiating process in some defined sense, as well as initiating an entirely independent process.

4.19.2. Deadlock avoidance is an issue that must be addressed for the higher level PTI facilities. This is an area for more research.

## 4.20. Mixed language working

4.20.1. A PTI should support the development of software in a variety of languages. Tools suitable for import to a PTI may be also written in a wide variety of languages. It is therefore important that a PTI is not restricted to any specific language either for its tools or for its applications. It must be possible for tools written in different languages to work together.

4.20.2. Mixed language working within programs is desirable because existing procedures may be available in a language different from the main language in which a program is being written.(e.g. programs written in Ada but using C procedures and vice versa). In the implementation of large systems it is unlikely that a single language will be suitable for everything. Implementors should be free to choose the best language for the job, knowing that mixed language working is supported.

4.20.3. There may be problems if some of the data types supported by different languages prove incompatible. If a facility is implemented only in one language there may be constraints on its use, implied by the implementation of the language and its associated loaders. For example if dynamic linking is available to C programs but not to Ada then it could be difficult to use an Ada facility with a C program and still use the C dynamic loader. This problem may be alleviated by the use of a common data representation.

## 4.21. Uniformity

4.21.1. Problems arising from different data representations in different language implementations could be significantly reduced by the use of standard data representations for all the data types in the supported languages, as well as standard representations for basic data structures such as separately compiled units. It would help in the support of language independent tools for debugging, linking and other functions so often supported by a different tool for each language.

4.21.2. If different languages represent data differently there will be problems for interworking of facilities written in different languages as well as difficulty with language independent tools. If an Ada compiler represents a vector of integers in one way and a C compiler uses a different representation then there must be some transformation from one to the other if facilities in the different languages are to work together. If tools are to communicate they need a common understanding of any shared data. This would be greatly assisted by the use of a common data representation. Although it would almost certainly prove uneconomic to alter the data representations associated with existing tools such as imported compilers, it would be possible to provide tools to transform the wide variety of data representations used in current tools to the target representation to enable language independent tools to work on the results. This would reduce the number of necessary transformations for inter-tool working from order($n^2$) to order($n$).

## 4.22. Distribution.

4.22.1. Distribution of a PTI across a network of user workstations is necessary. However this will pose problems for integrity and for performance unless care is taken in the specification of facilities which are known to be difficult in a distributed system. The facilities specification should be careful only to require facilities which are known to be feasible with current technology.

4.22.2. Problems of sharing data between workstations with different word length must be considered. A common system of data representation on the network may assist in this.

4.22.3. Filestore integrity in a distributed system is difficult unless a primitive is available for a distributed commit operation. If a transaction mechanism is provided in which commital of a transaction is to be atomic it must be clear what this means in terms of simultaneously committing amendments to several physically separate devices. If cyclic structures in a database may span several physically separate volumes there may be problems, should one volume fail while the others commit the update. The problems of detection of and recovery from filestore corruption caused by hardware or software failure during a psuedo atomic update which is not in fact physically atomic are considerable.

4.22.4. The PTI will need facilities to identify individual resources in the physical network and to communicate with specific resources.

4.22.5. The PTI needs to be able to handle changes to the network of physical resources, coping with failure of processors or devices without compromising integrity and permitting new devices and processors to be introduced into the network.

4.22.6. The PTI should be capable of operating usefully even if some elements of the network of physical resources are unavailable. It should therefore be possible to implement the PTI so that partitions of the network can be used in isolation.

4.22.7. Communication between resources requires a remote procedure call facility (refs 25 and 26) in additon to a message passing mechanism. There are problems with locking if a unit is locked from a remote processor which then goes down. A resilient lock mechanism across the network would help with this problem, for example as in SUN NFS (refs 27 and 28). Asynchronous distributed file accesses would also be most desirable (e.g. as provided in Decnet). Facilities are needed to implement a distributed read/modify/write as an atomic operation.

4.22.8. It may be desirable, in addition to providing a local identification for processes and data, to provide a system wide mechanism for identifying processes and data, independently of the allocation to the physical resources in the network.

## 4.23. Established technology

4.23.1. In order that the products meeting the PTI requirement can be developed and used with minimum risk it is desirable that each facility be based on a proven technology. The risk that a PTI product will not perform as required will be significantly increased if any facility depends on unproven technology. In addition, the acceptability of a product to potential users, (particularly to MoD project managers), will be increased if the product is perceived to be based on well established technology.

4.23.2. This requirement must not hinder research into new technologies. Some desirable characteristics for PTIs have to be omitted from the requirement because the technology to support them is not yet established. For example, there is no requirement to provide a distributed database because this is not yet possible using proven technology. As shown earlier the requirement to use established technology also implies the need for a kernel of low level facilities.

## 4.24. Technological compatibility

Where standards have already been established, for example the ISO and ANSI standards, they should be adopted. The standards are established technology and import of tools employing existing standards is desirable. Where standards are under development (for example X-windows, (refs 15 and 30)) the PTI should be designed as far as possible so as not to preclude their support in due course. The facilities specification should indicate which standards are to be supported and by what facilities.

## 4.25. PTI validation

4.25.1. It is important that the specification of a PTI be complete and unambiguous so that users can rely on the same interpretation of the specification in the different implementations. It is unlikely that a complete and consistent implementation specification will be achieved in the absence of a formal specification.

4.25.2. Each implementation of the PTI will need to be validated to check that it fully conforms to the specification. Such work will be hampered if there are ambiguities and gaps in the specification. Confidence in the validation process will be enhanced if a validation suite, based on a formal implementation specification of the PTI, is used to test individual implementations for conformance to the specification.

4.25.3. If users are to rely on all implementations behaving in the same way, then the implementation specification of a PTI should include not only the syntax for the various procedure calls in the selected language or languages but also the semantics of the interface.

4.25.4. A PTI may be implemented on a variety of different architectures and it should not be up to the implementors to make arbitrary or implementation dependent decisions. In the event that ambiguities or omissions are discovered in the PTI implementation specification, every effort should be made to enhance the specification in step with implementation decisions and to clarify the intended or correct interpretation of the specification.

## 4.26. Kernel facilities

4.26.1. It has been shown that many of the PTI requirements demand the provision of a kernel giving a good mapping to the underlying hardware. The quality of the kernel is central to the success of any PTI and the aim should be to keep it simple and as small as possible. It should include only those facilities without which an efficient and secure system cannot be written. All the higher level facilities must be implementable in terms of kernel facilities. The set of facilities needed in the kernel needs careful consideration. The syntax and semantics of the kernel facilities should aim to be machine and operating system independent. Any unavoidable dependencies should be isolated and parameter controlled. Some of the facilities needed are discussed below. This is a difficult area and the authors make no claim that the list is complete. It is intended to indicate the type of facilities to be considered.

4.26.2. For tool conversion the kernel needs to provide facilities analogous to those provided by current operating systems, although such facilities should be provided in a machine independent way as far as possible.

4.26.3. The need for safe and efficient interaction with filestore probably implies the need for the facilities to implement an efficient commit operation, support for the control of the mapping of data onto filestore and for controlling the order of interactions with filestore (necessary for the implementation of atomic operations on filestore; the commit operation must be sure that all the operations to be committed have completed before completing the commit). Facilities for mapping data onto filestore will also support fine granularity of data, both for efficient use of fine grained data structures and to provide for flexibility of data typing of values held on filestore.

4.26.4. One possibility is a facility to create directly addressable files and access them directly (without buffering) with random addressing. Constraints on contiguity and location of files, and a facility to extend a file would be useful additions.

4.26.5. Access control facilities are needed to guard against both malicious and accidental misuse. Protection against malicious interference is mandatory at least at the level of objects such as databases, database partitions, files and directories; these would probably be provided in terms of passwords and checks on appropriate use. The need to create and handle fine-grained database structures involving links between small values within these larger constructs implies a need for a system wide mechanism for creating, and for retaining on filestore, pointers to individual filestore values. Ideally, these pointers should be afforded the same degree of protection as the grosser structures; however this is unlikely to be achievable without an overall capability mechanism or system-wide type structure. It would probably be sufficient to guard against accidental misuse by making the pointers difficult to forge, provided that they cannot point outside some larger value which is protected against malice. The pointers could hold other information about the value to assist in access control, such as the type of use to which the value may be put (read/write or execute) and the size of the value.

4.26.6. Facilities for unanticipated highly interactive use, for passing data between concurrent processes and for sharing program code without privilege all imply a need for operations permitting the sharing of mainstore between processes. Interactive use also implies a need to perform dynamic linking.

4.26.7. Concurrent processing implies a need for monitors or other facilities for locking shared values, both in mainstore and in filestore. For efficiency locking must be possible on fine grained values, for example at the record rather than the file level. Multi-tasking requires kernel facilities for the initiation and control of both synchronous and asynchronous concurrent tasks, facilities for task synchronisation and for communication between tasks.

4.26.8. Distributed systems require facilities for remote procedure calls, message passing between processors and between tasks running concurrently in a single processor and for a resilient locking mechanism.

4.26.9. Interaction with other devices requires facilities both for synchronous and asynchronous devices.

## 5. Conclusion

5.1. This report gives a requirements specification for a portable public tools interface. The rationale not only discusses the reasons for the individual requirements but indicates that the requirements can largely be met. Although no one operating system or PTI currently satisfies the whole requirement, many individual bits of the requirement are demonstrated in different exisiting systems. It is, perhaps, a surprising conclusion that this requirement is not necessarily over ambitious.

5.2. One of the major conclusions must be the importance of establishing the right basis for a PTI. The low level facilities are crucial because everything else, the importing of existing tools, the performance and integrity both of tools and of the underlying system and the productivity of interactive users all rely on provision of an adequate core of low level facilities. Higher levels can be developed only if adequate kernel facilities are accessible.

5.3. Other major conclusions are the importance of a common data representation across the supported languages and the need for strong data typing. A common data representation helps in the solution of a variety of problems such as data and tool portability, interactive working and the provision of common tools for a variety of different languages. Strong data typing is essential for the safe composition of tools, for robustness and extensibility.

5.4. It is hoped that this requirement will be influential in shaping the requirement and facilities specifications for future PTI developments. The authors suggest that it could also be of use in evaluating and comparing PTI implementations. If PTIs are to be compared it is desirable that they all be measured against the same yardstick. This requirement would form a suitable basis for such a yardstick. Existing and proposed PTIs such as PCTE, PCTE+, CAIS (refs 18,22 and 23) and Ten15 (ref 29) should be measured against this requirement.

## 6. Acknowledgements

The authors wish to acknowledge their indebtedness to the RAC and to the draft EURAC documents in drawing up the requirements. Discussions with the IEPG study team responsible for the EURAC and consideration of PCTE1.4 has influenced the thinking. The authors also wish to acknowledge the help of their colleagues at RSRE, and the influence of current work on the Flex PSE and the Ten15 abstract machine on this work.

## 7. References


Ref 1. "On the security of Unix" by D.Ritchie, Bell Labs.

Ref 2. "Unix operating system security" by F.T.Grampp and R.H.Morris Bell Labs Technical Journal, Vol63 no8 Oct 84

Ref 3. DoD Requirements and Design Criteria for the Common APSE Interface set . 4 Oct 1986

Ref 4. PCTE Functional Specifications 4th edition

Ref 5. "A Review of the 1986 UniForm Panel: A secure Unix system and implications on CAIS and RAC" by H.Fischer Feb 86.

Ref 6. "A critique of Unix" by G.Blair, J.R.Malone and J.A.Mariani, Software Practice and Experience, Vol 15 no 12, Dec 85.

Ref 7. "Adam: an abstract database machine" by N.E.Peeling, J.D.Morison and E.V.Whiting. RSRE Report 84007, May 84.

Ref 8. "A Database approach to design data management and programming support for Ella, A high level HDDL" by N.E.Peeling and J.D.Morison, Proc. Computer Hardware Description Languages and their Applications, Tokyo, Japan 85

Ref 9. CAD Interface Standardisation Project (CISP) (Copyright 1984 with Crown, British Telecommunications, Ferranti Computer Systems, Ferranti Electronics Ltd, GEC PLC, ICL, Plessey Research Ltd, Racal Microelectronics Ltd, STL Ltd)

Ref 10. "Extending data typing beyond the bounds of programming languages" by M.Stanley, RSRE Memorandum 3878 1985

Ref 11. "Integrity and the Flex PSE" by M.Stanley, RSRE Memorandum 3915 1985

Ref 12. "An evaluation of the Flex PSE" by M.Stanley RSRE Report 86003 1986

Ref 13. "Reflections on some recent widespread computer break-ins" by B.Reid Communications of the ACM, Vol 30 No. 2,Feb 87.

Ref 14. "Prototyping a project master database for software engineering environments" by M.H.Penedo, TRW Inc. 2nd ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, Dec 86.

Ref 15 "Window system architectures -an overview" by P.Fritzson, Tutorial on Software Development Environments, Sweden, Nov 86.

Ref 16. "SDS Users Handbook" by SSL

Ref 17. "User assessment of Software tools, SDS. Report no 1 on Software Development System" by Scicon Ltd under contract to DoI, Nov 1982

Ref 18. Proposed MIL-STD-CAIS (31 Jan 85)

Ref 19. "Application Briefs" Computer Graphics World 8/82

Ref 20 "NS: An integrated symbolic design system" by J.Cherry, N.Mayle, C.Baker, H.Minsky, K Reti and N.Weste, Proc. IFIP International Conference, VLSI 85, Tokyo Aug 1985.

Ref 21 "The Siclops silicon compiler" by T.Hedges, K.Slater, G.Clow and T.Whitney, Proc. IEEE International Conference on Circuits and Computers, ICCC 82, New York Sept 1982.

Ref 22 "Recent developments in tool support interfaces. CAIS and PCTE" by T.Lyons and M.D.Tedd, 6th Ada UK Conference,

York Jan 1987.
Ref 23 "Technical overview of PCTE and CAIS" by T.Lyons and M.D.Tedd, 6th Ada UK Conference, York Jan 1987.
Ref 24 "On understanding types, data abstraction and polymorphism" by L.Cardelli and P.Wegner, ACM Computing Surveys Dec 85, Vol17 No 4
Ref 25 "Implementing remote procedure calls" by A.D.Birrell and B.J.Nelson, ACM Transactions on Computer Systems Feb 84, Vol2 No 1.
Ref 26. "Heterogeneous computing environments: report on the ACM SIGOPS workshop on accomodating heterogeneity" by D.Notkin, N.Hutchinson, J.Sanislo & M.Schwartz, Communications of the ACM, Feb 1987, Vol 30 No 2.
Ref 27. "Status monitor provides Network Locking Service for NFS" by JoMei Chang, Sun Microsystems report 1985
Ref 28. "Design and implementation of the SUN Network File System" by R.Sandberg, D.Goldberg, S.Kleiman, D.Walsh and B.Lyon Sun Microsystems report 1986
Ref 29. "Ten15: an overview" by P.W. Core and J.M.Foster " RSRE Memorandum 3977 1986.
Ref 30. "Xlib -C language X Interface, Protocol Version 10" by J.Gettys, R.Newman and T.Della Fera, MIT Project Athena, 1986.
Ref 31 "The Siclops silicon compiler" by T.Hedges, K.Slater, G Clow and T.Whitney, Proc. IFIP International Conference, VLSI 85, Tokyo Aug 1985.

## 8. Glossary

Ada A trademark of the Ada Joint Program Office.
APSE Ada Project Support Environment
AT&T the developers of Unix
C A programming language.
CAIS Common APSE Interface Set -tools interface set for Ada project support environments
CNAD Committe of National Armament Directors
Dynamic linking -the ability to load modules at run time rather than running a statically linked system that combines separately compiled program components into an amorphous code image.
EURAC European Requirements and Design Criteria for a public tools interface set
George 3 -operating system for the ICL 1900 series of computers.
IEPG Independent European Programme Group - a European/NATO industry and government group fostering collaboration on European defence work
IPR Intellectual Property Rights
NRAC NATO Requirements and Design Criteria for a public tools interface set
OMS Object Management System: an entity relationship database which forms the core of PCTE.
PACT a project under a contract from CEC ESPRIT programme to

develop tools for PCTE.

PCTE A Basis for a Portable Common Tools Environment: a tools interface set developed by Bull SA; The General Electric Co. plc;ICL; Nixdorf Computer AG; Olivetti SPAand Siemens AG under a contract from CEC ESPRIT programme.

PCTE1.4 Version 1.4 of the PCTE functional specification

Posix- Portable Operating System, a trademark of the US Institute of Electronic and Electrical Engineers (IEEE). Posix is being promoted in the USA as a possible standard. It is a multi-user, multi-tasking operating system based on Unix V, release 3.0.

PTI Public Tools Interface

RAC DoD Requirements and Design Criteria for the Common APSE Interface set

SD Systems Designers plc.

SDS Software Development System- a database system developed jointly by RSRE and SSL for use in development of large software systems.

SSL Software Sciences Ltd.

TSGCEE SWG on Ada/APSE Tri Service Group on Communications and Electronic Equipment, Special Working Group on Ada Programming Support Environments, a NATO study group set up by the TSGCEE.

Unix A trademark of Bell Laboratories, AT&T

Unix BSD4.2 Unix Version 4.2 from the University of California at Berkeley.

Unix V. Version V of Unix.

VDM The Vienna Development Method- a notation for writing formal specifications

X/OPEN Portability Guide - a guide published by the X/OPEN group of European companies on the standardisation of Unix for business users, published 1985 by North Holland, Elsevier. ISBN 0-444-87839-4

DRIC T CONTROL SHEET

Overall security classification of sheet UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S) )

| 1. DRIC Reference (if known) | 2. Originator's Reference | 3. Agency Reference | 4. Report Security Classification |
|---|---|---|---|
| | R3TCC4 | | U/C |

| 5. Originator's Code (if known) | 6. Originator (Corporate Author) Name and Location |
|---|---|
| 77S [illegible] | RSRE SAINT ANDREWS ROAD MALVERN WORCS WR14 3PS |

| 6a. Sponsoring Agency's Code (if known) | 6a. Sponsoring Agency (Contract Authority) Name and Location |
|---|---|
| | |

7. Title Public tool interfaces for software development environments

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

| 8. Author 1 Surname, initials | 9(a) Author 2 | 9(b) Authors 3,4... | 10. Date | pp. ref. |
|---|---|---|---|---|
| STANLEY M | PEELING NE | CURRIE IF | 1987.09 | 35 |

| 11. Contract Number | 12. Period | 13. Project | 14. Other Reference |
|---|---|---|---|
| | | | |

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

See text

END

DATE
FILMED

3-88

DTIC